APPLICATION FOR UNITED STATES LETTERS PATENT

For

BIOS FRAMEWORK FOR ACCOMMODATING MULTIPLE SERVICE PROCESSORS ON A SINGLE SERVER TO FACILITATE DISTRIBUTED/SCALABLE SERVER MANAGEMENT

Inventors:

Rahul Khanna
Mallik Bulusu
Vincent J. Zimmer

Prepared by:

BLAKELY SOKOLOFF TAYLOR & ZAFMAN LLP
12400 Wilshire Boulevard
Los Angeles, CA 90025-1026
(206) 292-8600

Attorney's Docket No.: 42P18122

"Express Mail" mailing label number: EV320120147US
Date of Deposit: March 29, 2004
I hereby certify that I am causing this paper or fee to be deposited with the United States Postal Service "Express Mail Post Office to Addressee" service on the date indicated above and that this paper or fee has been addressed to the Mail Stop Patent Application, Commissioner for Patents, PO Box 1450, Alexandria, VA 22313-1450

Adrian Villarreal
(Typed or printed name of person mailing paper or fee)

(Signature of person mailing paper or fee)

March 29, 2004
(DATE SIGNED)

BIOS FRAMEWORK FOR ACCOMMODATING MULTIPLE SERVICE PROCESSORS ON A SINGLE SERVER TO FACILITATE DISTRIBUTED/SCALABLE SERVER MANAGEMENT

FIELD OF THE INVENTION

[0001] The field of invention relates generally to computer systems and, more specifically but not exclusively relates to a BIOS framework for accommodating multiple service processors on a single server to facilitate distributed/scalable server management.

BACKGROUND INFORMATION

[0002] As computer server architectures have advanced, more specific functionality have been added to meet customer needs and to increase uptime. For example, older computer server architectures might employ a single processor that is use to provide substantially all server functionality via execution of firmware and software instructions on the processor, as well as through specific hardware-level logic built into the processor and/or platform. More recently, the single point of service has been discarded for a more distributed service scheme, whereby multiple processors are employed to perform targeted functions.

[0003] For example, modern servers may employ an "out-of-band" management controller that performs separate functions than the servers' primary processor (or processors for multi-processor platforms). Typically, an out-of-band management controller comprises an independent processor, such as a base management controller (BMC) or service processor, connected to various hardware components of a server platform to monitor the functionality of those hardware components. For instance, a service processor may be configured to have its own independent link to a network with an independent Internet protocol ("IP") address to allow an administrator on a remote console to monitor the functionality of the server. As used herein, these processors are collectively termed "service processors."

[0004] With reference to Figure 1, a server 100 having a conventional service processor configuration known in the art is depicted. The illustrated embodiment of server 100 includes a service processor 102, a main processor (CPU) 104, a communication interface 106, a data storage unit 108, a service processor firmware storage device 110, and a platform firmware storage device 112. Main processor 104 is communicatively coupled to various platform components via one or more buses that are collectively illustrated as a system bus 114. Typically, service processor 102 is coupled to the same and/or different platform components via an independent bus and/or direct channels, also called service channels; this bus or buses is depicted in Figure 1 as a management bus 116. In one embodiment, service processor 102 is communicatively-coupled to communication interface 106 via a separate channel 118. Optionally, the coupling may be implemented via management bus 116.

[0005] Generally, service processor 102 may be linked in communication with a network 120 via either communication interface 106 or a dedicated network interface. In the illustrated embodiment, communication interface 106 provides two ports with respective IP addresses of IP_1 and IP_2, whereby one IP address may be used by main processor 104, while the other may be used by service processor 102

[0006] An administrator working on a remote console 120 coupled to network 122 can monitor the functionality of main processor 104, data storage unit 108, or other entities (not shown) via interaction with service processor 102. The functions of service processor 102 generally include monitoring one or more characteristics or operations of main processor 104 (e.g., monitoring the temperature of processor 104), data storage unit 108, and other hardware components (not shown), recording hardware errors, performing manual tasks initiated by the administrator (such as resetting main processor 104), recovering main processor 104 after an

error, performing manual input/output data transfers, and the like. The functions are collectively depicted as services 124

[0007] The foregoing service processor functions are enabled via execution of firmware stored in service processor firmware storage device 110. In particular, interaction with the various hardware components is provided via one or more corresponding firmware drivers. At the same time, separate firmware drivers stored in platform firmware storage device 112 are employed by main processor 104 to access the same hardware components.

[0008] While the conventional scheme supports the potential of a wide-range of services, it isn't scalable. This is problematic. For example, the choice of service processor (and associated firmware) for a given platform design is always a challenge, as the server management requirements vary drastically from customer to customer. One may want to employ a light-weight service processor to minimize costs. On the other hand, a customer may want to opt for a full-blown BMC implementation on high-end servers. It would be advantageous to have a flexible and scalable service management solution to mitigate the foregoing limitations with conventional service management implementation.

BRIEF DESCRIPTION OF THE DRAWINGS

[0009] The foregoing aspects and many of the attendant advantages of this invention will become more readily appreciated as the same becomes better understood by reference to the following detailed description, when taken in conjunction with the accompanying drawings, wherein like reference numerals refer to like parts throughout the various views unless otherwise specified:

[0010] Figure 1 is a schematic diagram of a conventional server architecture that includes support for implementing a single service processor;

[0011] Figure 2 is a schematic diagram of a scalable server management framework that includes support for concurrent implementation of multiple service processors, and provides a unified presentation of service capabilities to service consumers, according to one embodiment of the invention;

[0012] Figure 3 is a flowchart illustrating operations and logic performed during a pre-boot phase of a server having the architecture of Figure 2 to setup up a BIOS unified presentation table and publish a BIOS-based server management handler, according to one embodiment of the invention;

[0013] Figure 4 is a is a schematic diagram illustrating the various execution phases that are performed in accordance with the extensible firmware interface (EFI) framework under which the operations of the platform initialization process of Figure 3 may be performed, according to one embodiment of the invention;

[0014] Figure 5 is a block schematic diagram illustrating various components of the EFI system table corresponding to the EFI framework;

[0015] Figure 6 is a schematic diagram illustrating further details of how services are enumerated and published under the EFI framework;

[0016] Figure 7 is a schematic diagram of a server having an architecture based on the scalable server management framework of Figure 2 that is enabled to provide

data to render a unified presentation of service capabilities on a remote control used by an administrator to request and observer server management services.

[0017] Figure 8 is a representation of a user interface rendered on the remote console that enables an end-user to make preferences identifying a usage order of service processors that support the same service; and

[0018] Figure 9 is a flowchart illustrating operations performed during a server management event that is serviced by selecting an appropriate service processor, according to one embodiment of the invention.

DETAILED DESCRIPTION OF PREFERRED EMBODIMENTS

[0019] Embodiments of a BIOS framework for accommodating multiple service processors on a single server to facilitate distributed/scalable server management are described herein. In the following description, numerous specific details are set forth to provide a thorough understanding of embodiments of the invention. One skilled in the relevant art will recognize, however, that the invention can be practiced without one or more of the specific details, or with other methods, components, materials, *etc.* In other instances, well-known structures, materials, or operations are not shown or described in detail to avoid obscuring aspects of the invention.

[0020] Reference throughout this specification to "one embodiment" or "an embodiment" means that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment of the present invention. Thus, the appearances of the phrases "in one embodiment" or "in an embodiment" in various places throughout this specification are not necessarily all referring to the same embodiment. Furthermore, the particular features, structures, or characteristics may be combined in any suitable manner in one or more embodiments.

[0021] Throughout this specification and in the claims, several terms of art are used. These terms are to take on their ordinary meaning in the art from which they come, unless specifically defined herein (such as follows) or the context of their use would clearly suggest otherwise. BIOS (Basic Input-Output System) refers to the system firmware (i.e., executable instructions) that is employed to facilitate low-level interfacing with platform hardware. As used herein, the terms "BIOS" and "firmware" may be used interchangeably.

[0022] The use of the term "service processor" and services supported by the service processor are often implied herein. However, it will be understood that a

service processor does not perform services by itself, but rather performs services via execution of "service code" associated with the service processor.

[0023] In accordance with aspects of the embodiments described herein, a BIOS-based framework that facilitates scalability of server management operations is disclosed. The framework enables various server management components, such as commercial off-the-shelf (COTS) server management hardware solutions, to be added to servers to achieve scalability. To accommodate distributed/scalable server management, the framework generates a BIOS Unified Presentation (BUP) of overall server management service capabilities available to consumers of server management services. The framework enables server management components to be added, removed and/or replaced, with any new or removed service capabilities supported by new platform configurations reflected by updating the BUP. The framework also provides a single point interface to all consumers of server management services.

[0024] An overview illustrating a scalable server management framework 200 according to one embodiment is shown in Figure 2. In one embodiment of the framework, a main processor 202 is communicatively-coupled to one or more service processors via a management bus or the like. In the illustrated embodiment, these include service processors 204, 206, and 208. (For convenience, these service processors are labeled SM1, SM2, and SM3, respectively.) In one embodiment, service processor 204 comprises a BMC. In one embodiment, a service management processor may perform the interface functions described below that are depicted in Figure 2 as being performed by main processor 202.

[0025] In general, a typical server implementation may include one service processor, such as a BMC, that is built into the main or baseboard of the server. Additional service processors, such as depicted by service processors 206 and 208, are provided by add-in cards that are connected to expansion slots provided by the

baseboard or communicatively-coupled via a shared backplane or the like. As such, these service processors are also referred to herein as "add-in" service processors. In some embodiments, the server may be configured as a blade server, which may include one or more shared backplanes.

[0026] Each of the service processors provides a set of services via execution of corresponding service code. The services shown for service processors 204, 206, and 208 are depicted as service sets 210, 212 and 214, respectively. In one embodiment, each service is facilitated by a corresponding firmware component (i.e., set of instructions comprising service code) that is executed by the service processor hosting the service. In the illustrated embodiment of Figure 2, this firmware is depicted as BMC processor firmware 216, add-in service processor firmware 218, and add-in service processor firmware 220.

[0027] In one embodiment, the BUP functionality is facilitate by a firmware component depicted as BUP firmware 222. In the illustrated embodiment, this firmware component is stored as part of the platform's system BIOS 224. As described below, the BUP firmware may be stored elsewhere on the baseboard, or may be loaded from an add-in card or even from a network store.

[0028] The BUP firmware maintains a BUP table 226. The BUP table maps an aggregated set of services provided by the various service processors present in a system to services offered by those service processors. This supports a unified presentation of the service capabilities to server management service consumers for the system. In one embodiment, BUP-related information is provided to such consumers via consumers to server management infrastructure 228, which represents an abstraction of all of the interfaces that enable server management service consumers to access the service facilities for the system.

[0029] Under the framework illustrated in Figure 2, firmware facilities are implemented to manage the BUP and handle interfacing with service consumers.

To support this functionality, the service capabilities for each service processor are collected by BUP firmware 222. In one embodiment, this is accomplished via a service processor registration process. Under the process, a firmware driver for each service processor registers its instance to the BUP during the initialization phase for the driver. Each service processor also publishes its service capabilities to the BUP via corresponding interfaces. Through this mechanism, the BUP captures an active list of service processors and their service capabilities (via execution of associated service code) at any given time.

[0030] A general overview of a service processor registration process, according to one embodiment, is now presented with reference to the flowchart of Figure 3. Further details of a more-specific implementation of this process depicted in Figures 4, 5, and 6 follows.

[0031] Referring to Figure 3, the registration process begins with a power-on event (or system reset), as depicted in a block 300. In response, firmware initialization operations are performed during a pre-boot phase to verify the operational integrity of the system and prepare the system for loading an operating system. Included in this process is initialization of the BUP framework in a block 302. The next set of nested operations are performed for each service processor and each firmware driver for that service processor, as depicted by outside loop start and end loop blocks 304 and 318, and inside loop start and end loop blocks 306 and 316.

[0032] In a block 306 each firmware driver is loaded and/or executed during the pre-boot phase of the server. This includes registering an instance of each firmware driver. Depending on the particular driver, the corresponding firmware may be used only for initialization, or may set up interfaces for use during operating system (OS) runtime. As indicating by a decision block 310, if the driver is a service processor driver, the logic proceeds to a block 312. In this block, the services supported via

the service processor driver are enumerated. The enumerated services are then published to the BUP. The operations of blocks 308, 310, 312, and 314 are repeated until all the firmware drivers for the system have been loaded and/or executed.

[0033] Subsequently, a BIOS server management handler is established in a block 320. The handler is used to provide an interface to server management consumers, as described in further detail below. At the close of the registration process, the pre-boot initialization of the system continues, with the operating system being booted in a block 320.

[0034] In accordance with one embodiment, the foregoing service processor registration process may be implemented under an extensible firmware framework known as the Extensible Firmware Interface (EFI) (specifications and examples of which may be found at http://developer.intel.com/technology/efi). EFI is a public industry specification that describes an abstract programmatic interface between platform firmware and shrink-wrap operation systems or other custom application environments. The EFI framework include provisions for extending BIOS functionality beyond that provided by the BIOS code stored in a platform's BIOS device (*e.g.*, flash memory). More particularly, EFI enables firmware, in the form of firmware modules and drivers, to be loaded from a variety of different resources, including primary and secondary flash devices, option ROMs, various persistent storage devices (*e.g.*, hard disks, CD ROMs, etc.), and even over computer networks. The current EFI framework specification is entitled, "Intel Platform Innovation for EFI Architecture Specification," version 0.9, September 16, 2003.

[0035] Figure 4 shows an event sequence/architecture diagram used to illustrate operations performed by a platform (e.g., server) under the EFI framework in response to a cold boot (*e.g.*, a power off/on reset). The process is logically divided into several phases, including a pre-EFI Initialization Environment (PEI) phase, a

Driver Execution Environment (DXE) phase, a Boot Device Selection (BDS) phase, a Transient System Load (TSL) phase, and an operating system runtime (RT) phase. The phases build upon one another to provide an appropriate run-time environment for the OS and platform.

[0036] The PEI phase provides a standardized method of loading and invoking specific initial configuration routines for the processor (CPU), chipset, and motherboard. The PEI phase is responsible for initializing enough of the system to provide a stable base for the follow on phases. Initialization of the platforms core components, including the CPU, chipset and main board (i.e., motherboard) is performed during the PEI phase. This phase is also referred to as the "early initialization" phase. Typical operations performed during this phase include the POST (power-on self test) operations, and discovery of platform resources. In particular, the PEI phase discovers memory and prepares a resource map that is handed off to the DXE phase. The state of the system at the end of the PEI phase is passed to the DXE phase through a list of position independent data structures called Hand Off Blocks (HOBs).

[0037] The DXE phase is the phase during which most of the system initialization is performed. The DXE phase is facilitated by several components, including the DXE core 400, the DXE dispatcher 402, and a set of DXE drivers 404. The DXE core 400 produces a set of Boot Services 406, Runtime Services 408, and DXE Services 410. The DXE dispatcher 402 is responsible for discovering and executing DXE drivers 404 in the correct order. The DXE drivers 404 are responsible for initializing the processor, chipset, and platform components as well as providing software abstractions for console and boot devices. These components work together to initialize the platform and provide the services required to boot an operating system. The DXE and the Boot Device Selection phases work together to establish consoles and attempt the booting of operating systems. The DXE phase is

terminated when an operating system successfully begins its boot process (*i.e.*, the BDS phase starts). Only the runtime services and selected DXE services provided by the DXE core and selected services provided by runtime DXE drivers are allowed to persist into the OS runtime environment. The result of DXE is the presentation of a fully formed EFI interface.

[0038] The DXE core is designed to be completely portable with no CPU, chipset, or platform dependencies. This is accomplished by designing in several features. First, the DXE core only depends upon the HOB list for its initial state. This means that the DXE core does not depend on any services from a previous phase, so all the prior phases can be unloaded once the HOB list is passed to the DXE core. Second, the DXE core does not contain any hard coded addresses. This means that the DXE core can be loaded anywhere in physical memory, and it can function correctly no matter where physical memory or where Firmware segments are located in the processor's physical address space. Third, the DXE core does not contain any CPU-specific, chipset specific, or platform specific information. Instead, the DXE core is abstracted from the system hardware through a set of architectural protocol interfaces. These architectural protocol interfaces are produced by DXE drivers 104, which are invoked by DXE Dispatcher 102.

[0039] The DXE core produces an EFI System Table 500 and its associated set of Boot Services 406 and Runtime Services 408, as shown in Figure 5. The DXE Core also maintains a handle database 502. The handle database comprises a list of one or more handles, wherein a handle is a list of one or more unique protocol *GUIDs* (Globally Unique Identifiers) that map to respective protocols 504. A protocol is a software abstraction for a set of services. Some protocols abstract I/O devices, and other protocols abstract a common set of system services. A protocol typically contains a set of APIs and some number of data fields. Every protocol is named by a GUID, and the DXE Core produces services that allow protocols to be registered in

the handle database. As the DXE Dispatcher executes DXE drivers, additional protocols will be added to the handle database including the architectural protocols used to abstract the DXE Core from platform specific details.

[0040] The Boot Services comprise a set of services that are used during the DXE and BDS phases. Among others, these services include Memory Services, Protocol Handler Services, and Driver Support Services: Memory Services provide services to allocate and free memory pages and allocate and free the memory pool on byte boundaries. It also provides a service to retrieve a map of all the current physical memory usage in the platform. Protocol Handler Services provides services to add and remove handles from the handle database. It also provides services to add and remove protocols from the handles in the handle database. Addition services are available that allow any component to lookup handles in the handle database, and open and close protocols in the handle database. Support Services provides services to connect and disconnect drivers to devices in the platform. These services are used by the BDS phase to either connect all drivers to all devices, or to connect only the minimum number of drivers to devices required to establish the consoles and boot an operating system (*i.e.*, for supporting a fast boot mechanism). In contrast to Boot Services, Runtime Services are available both during pre-boot and OS runtime operations.

[0041] The DXE Services Table includes data corresponding to a first set of DXE services 506A that are available during pre-boot only, and a second set of DXE services 506B that are available during both pre-boot and OS runtime. The pre-boot only services include Global Coherency Domain Services, which provide services to manage I/O resources, memory mapped I/O resources, and system memory resources in the platform. Also included are DXE Dispatcher Services, which provide services to manage DXE drivers that are being dispatched by the DXE dispatcher.

[0042] The services offered by each of Boot Services 406, Runtime Services 408, and DXE services 410 are accessed via respective sets of API's 412, 414, and 416. The API's provide an abstracted interface that enables subsequently loaded components to leverage selected services provided by the DXE Core.

[0043] After DXE Core 400 is initialized, control is handed to DXE Dispatcher 402. The DXE Dispatcher is responsible for loading and invoking DXE drivers found in firmware volumes, which correspond to the logical storage units from which firmware is loaded under the EFI framework. The DXE dispatcher searches for drivers in the firmware volumes described by the HOB List. As execution continues, other firmware volumes might be located. When they are, the dispatcher searches them for drivers as well.

[0044] There are two subclasses of DXE drivers. The first subclass includes DXE drivers that execute very early in the DXE phase. The execution order of these DXE drivers depends on the presence and contents of an *a priori* file and the evaluation of dependency expressions. These early DXE drivers will typically contain processor, chipset, and platform initialization code. These early drivers will also typically produce the architectural protocols that are required for the DXE core to produce its full complement of Boot Services and Runtime Services.

[0045] The second class of DXE drivers are those that comply with the EFI 1.10 Driver Model. These drivers do not perform any hardware initialization when they are executed by the DXE dispatcher. Instead, they register a Driver Binding Protocol interface in the handle database. The set of Driver Binding Protocols are used by the BDS phase to connect the drivers to the devices required to establish consoles and provide access to boot devices. The DXE Drivers that comply with the EFI 1.10 Driver Model ultimately provide software abstractions for console devices and boot devices when they are explicitly asked to do so.

[0046] Any DXE driver may consume the Boot Services and Runtime Services to perform their functions. However, the early DXE drivers need to be aware that not all of these services may be available when they execute because all of the architectural protocols might not have been registered yet. DXE drivers must use dependency expressions to guarantee that the services and protocol interfaces they require are available before they are executed.

[0047] The DXE drivers that comply with the EFI 1.10 Driver Model do not need to be concerned with this possibility. These drivers simply register the Driver Binding Protocol in the handle database when they are executed. This operation can be performed without the use of any architectural protocols. In connection with registration of the Driver Binding Protocols, a DXE driver may "publish" an API by using the *InstallConfigurationTable* function. This published drivers are depicted by API's 418. Under EFI, publication of an API exposes the API for access by other firmware components. The API's provide interfaces for the Device, Bus, or Service to which the DXE driver corresponds during their respective lifetimes.

[0048] The BDS architectural protocol executes during the BDS phase. The BDS architectural protocol locates and loads various applications that execute in the pre-boot services environment. Such applications might represent a traditional OS boot loader, or extended services that might run instead of, or prior to loading the final OS. Such extended pre-boot services might include setup configuration, extended diagnostics, flash update support, OEM value-adds, or the OS boot code. A Boot Dispatcher 420 is used during the BDS phase to enable selection of a Boot target, *e.g.*, an OS to be booted by the system.

[0049] During the TSL phase, a final OS Boot loader 422 is run to load the selected OS. Once the OS has been loaded, there is no further need for the Boot Services 406, and for many of the services provided in connection with DXE drivers 404 via API's 418, as well as DXE Services 406A. Accordingly, these

reduced sets of API's that may be accessed during OS runtime are depicted as API's 416A, and 418A in Figure 4.

[0050] In accordance with some embodiments, the EFI pre-boot/boot framework of Figures 4 and 5 may be implemented to facilitate initialization and run-time support of the foregoing BUP server management functions. This is facilitated, in part, by API's published by respective components/devices during the DXE phase, and through use of the Variable Services runtime service, which is used to update BUP table entries in response to platform configuration changes.

[0051] For example, an exemplary scheme for initializing BUP server management facilities is shown in Figure 6. During the DXE phase, a DXE core server management driver 600 is loaded and executed. In accordance with the framework embodiment of Figure 2, firmware corresponding to core server management driver 600 comprises a portion of BUP firmware 222. As such, this firmware component is loaded from system BIOS 224.

[0052] In modern computer systems, the system BIOS is stored in a memory store called a "boot firmware device" (BFD). BFDs will typically comprise a rewritable non-volatile memory component, such as, but not limited to, a flash device or EEPROM chip. As used herein, these devices are termed "non-volatile (NV) rewritable memory devices." In general, NV rewritable memory devices pertain to any device that can store data in a non-volatile manner (*i.e.*, maintain data when the computer system is not operating), and provides both read and *write* access to the data. Thus, all or a portion of firmware stored on an NV rewritable memory device may be updated by rewriting data to appropriate memory ranges (e.g., blocks) defined for the device. Firmware may also be stored in NV memory devices, such as conventional ROMs (read-only memory).

[0053] In response to a system reset or power on event, the system performs pre-boot system initialization operations in the manner discussed above with

reference to Figure 3. Upon being reset, the processor executes reset stub code that jumps execution to the base address of the BFD (*e.g.*, a device hosting system BIOS 224) via a reset vector. The BFD contains firmware instructions that are logically divided into a boot block and an EFI core.

[0054] The boot block contains firmware instructions for performing early initialization, and is executed by processor 202 to initialize the CPU, chipset, and motherboard. (It is noted that during a warm boot early initialization is not performed, or is at least performed in a limited manner). Execution of firmware instructions corresponding to the EFI core are executed next, leading to the DXE phase. As part of initializing the DXE core is initialized, core server management driver 600 is loaded. In turn, this driver is used to initialize the BUP framework, as discussed above with referenced to block 302 of Figure 3.

[0055] Henceforth, DXE dispatcher 402 begins loading DXE drivers 404. Each DXE driver corresponds to a system component, and provides an interface for directly accessing that component. Included in the DXE drivers are drivers that will be subsequently employed for registering service processors and supporting OS-runtime server management operations. In Figure 6, these DXE drivers include a DXE driver 602, which is loaded from BMC processor firmware 216, and DXE drivers 604 and 606, which are loaded from add-in service processor firmware 218 and 220, respectively. Loading of DXE drivers 602, 604, and 604 causes corresponding API's 608, 610 and 612 to be published by the EFI framework. In one embodiment, data relating to the BUP is stored in a BUP table 508 of the EFI system configuration table (Figure 5).

[0056] Initially, a DXE driver corresponding to a primary service processor will be loaded, while DXE drivers corresponding to add-in service processors hosted by add-in cards will be subsequently discovered and loaded. In one embodiment, the service processor registration process supports dynamic registration. What this

means is that services provided via a "hot-swap" service processor add-in card may be published to the BUP framework, enabling the framework to present any new services offered by the add-in card in its unified list of services. In a similar manner, when a hot-swap add-in card is removed, its corresponding services are likewise removed from the unified list.

[0057] An illustration of a unified presentation of an exemplary set of services offered by various service processors (and associated service code) hosted by a server 700 having a configuration similar to the framework embodiment of Figure 2 is shown in Figure 7. In one embodiment, a BUP table 226 includes an aggregated list of services offered by all available service processors for server 700. For example, this would correspond to the left hand column of BUP table 226. In the illustrated embodiment, the BUP table further shows a grid of services vs. service processor. This enables an administrator or the like to select a particular service processor to perform a selected service. This is often advantageous, as it enables the administrator to load-balance the workload performed by the service processors for a given system.

[0058] In one embodiment, the administrator or similar end-user is enabled to set up use preferences, whereby a service processor having a higher preference among multiple service processors that support like services is selected to perform the service. For instance, Figure 7 shows a BUP 226A illustrating one embodiment of a service preference scheme. Under the scheme, an end-user is enabled to set a preferred order of service processors to perform a given task. For example, SERVICE A is supported by each of service processors SM1, SM2, and SM3 (i.e., service processors 204, 206, and 208). It is desired by the end-user to have service processor SM2 perform this task, if available. If service processor SM2 is unavailable, the preference falls to service processor SM3. If neither service

processor SM2 or SM3 is available, then service processor SM1 is assigned to perform the service.

[0059] In one embodiment, an end-user is enabled to set up preferences during pre-boot system initialization operations. For instance, an EFI application may be employed to present a text-based interface to an end-user of server 700 during its pre-boot phase. In another embodiment, use preferences may be entered during OS-runtime. In this instance, an EFI application or DXE driver is used to publish an API that is available for runtime services. In turn, the API enables a runtime component, such as a system management application to access (i.e., retrieve and/or manipulate) the BUP table data and display such information to an end-user via an appropriate user-interface. Depending on the implementation, the interface presented to the end-user may be either a text-based interface or a graphical user interface.

[0060] Figure 9 shows a flowchart illustrating operations performed during handling of a service management event, according to one embodiment. The processor begins in response to operations performed in a block 900, wherein a service consumer initiates a server management request. In general, a service consumer may comprise any entity that may request server management services to be performed on its behalf. This includes both humans (e.g., administrators) and programmatic entities (e.g., a software-based server management component). In instances in which the service consumer is an end-user, a software-based service host utility may be employed to provide the end-user with service availability and selection operations, along with corresponding information that is displayed while a service is being performed, such as progress, status, results, data dumps, etc. (not shown)

[0061] In response to the server management request, the BUP framework identifies one or more (as applicable) service processors that are capable of

servicing the request, as depicted in a block 902. If preferences are supported, the BUP framework further filters the selection process based on preferences set up by the end user (such as illustrated in Figure 8). In a block 906, the BUP framework broadcasts the server management request to the relevant service processor(s). In one embodiment in which preferences are not employed, the broadcast is used to access the first available service processor, thus the broadcast is made to all service processors. Under a preference-based scheme, the broadcast (or a unicast) may be targeted toward a selected service processor with the highest preference. The process is completed in a block 908, wherein the service processor(s) service the request and update the BUP of status, results, etc.

[0062] In the foregoing embodiments, firmware and software components are used to support the enhanced server management functions provided by the exemplary BUP framework implementations described herein. Thus, embodiments of this invention may be used as or to support a firmware and/or software executed upon some form of processing core (such as a service processor of a server) or otherwise implemented or realized upon or within a machine-readable medium. A machine-readable medium includes any mechanism for storing or transmitting information in a form readable by a machine (*e.g.*, a computer). For example, a machine-readable medium can include such as a read only memory (ROM); a random access memory (RAM); a magnetic disk storage media; an optical storage media; and a flash memory device, *etc.* In addition, a machine-readable medium can include propagated signals such as electrical, optical, acoustical or other form of propagated signals (*e.g.*, carrier waves, infrared signals, digital signals, *etc.*).

[0063] The above description of illustrated embodiments of the invention, including what is described in the Abstract, is not intended to be exhaustive or to limit the invention to the precise forms disclosed. While specific embodiments of, and examples for, the invention are described herein for illustrative purposes,

various equivalent modifications are possible within the scope of the invention, as those skilled in the relevant art will recognize.

[0064] These modifications can be made to the invention in light of the above detailed description. The terms used in the following claims should not be construed to limit the invention to the specific embodiments disclosed in the specification and the claims. Rather, the scope of the invention is to be determined entirely by the following claims, which are to be construed in accordance with established doctrines of claim interpretation.